Reconfigurable Accelerator with Binary Compatibility for General Purpose Processors

Antonio Carlos Schneider Beck, Luigi Carro

Universidade Federal do Rio Grande do Sul – Instituto de Informática
Av. Bento Gonçalves, 9500 – Campus do Vale – Porto Alegre/Brazil
{caco,carro}@inf.ufrgs.br

Abstract. Although transistor scaling keeps following Moore`s law, and more area is available for designers, the clock frequency and ILP rate do not present the same level of growth anymore. This way, new architectural alternatives are necessary. Reconfigurable fabric appears to be one emerging possibility: besides exploiting the parallelism among instructions, it can also accelerate sequences of data dependent ones. However, reconfiguration wide spread usage is still withheld by the need of special tools and compilers, which clearly do not sustain the reuse of legacy code without any kind of modification. Based on all these facts, this work proposes a new Binary Translation algorithm, implemented in hardware and working in parallel to the processor, responsible for transforming sequences of instructions at run-time to be executed on a dynamic coarse-grain reconfigurable array, tightly coupled to a traditional RISC machine. Therefore, we can take advantage of using pure combinational logic to optimize even control-flow oriented code in a totally transparent process, without any modification in the source code or binary. Using the Simplescalar Toolset together with the MIBench embedded benchmark suite, we show performance improvements and area evaluation when comparing against a traditional superscalar architecture.

Introduction

The possibility of increasing the number of transistors inside an integrated circuit with the passing years, following Moore´s Law, has been pushing performance at the same level of growth. However, high performance architectures as the diffused superscalar machines are now challenging well known limits of the ILP [1]: considering the Intel's family of processors, the IPC rate has not increased since the Pentium Pro [2]. This way, recent speed-ups in performance occurred mainly thanks to boosts in clock frequency through the employment of deeper pipelines. Even this approach, though, is reaching a limit. For example, the clock frequency of Intel's Pentium 4 processor only increased from 3.06 to 3.8 GHz between 2002 and 2006 [3].

Because of these reasons, companies are migrating to chip multiprocessors to take advantage of the extra area available, even though there is still a huge potential to speed up a single thread software. Hence, new architectural alternatives that can take

Please use the following format when citing this chapter:

Beck, A.C.S. and Carro, L., 2009, in IFIP International Federation for Information Processing, Volume 291; *VLSI-SoC: Advanced Topics on Systems on a Chip*; eds. R. Reis, V. Mooney, P. Hasler; (Boston: Springer), pp. 271–286.

advantage of the integration possibilities and that can address the performance issues stated before become necessary.

Reconfigurable fabric appears to be a serious candidate to be one of these solutions. By translating a sequence of operations into a combinational circuit performing the same computation, one could gain performance and reduce energy consumption at the price of extra area [4][5]. Furthermore, at the same time that reconfigurable computing can explore the ILP of the applications, it also speeds up sequence of data dependent instructions, which is its main advantage when comparing to traditional architectures. Dataflow architectures put this concept to the edge, achieving huge speed-ups [11].

Another advantage of reconfigurable architectures is their regularity: it is common sense that as the more the technology shrinks, the more important regularity becomes – since this will affect the reliability of printing the geometries employed today in 65 nanometers and below [6]. Besides being more predictable, regular circuits are also low cost, since as more customizable the circuit is, more expensive it becomes. This way, reconfigurable architectures based on regular fabric could solve the mask cost and many other issues such as printability, power integrity and other aspects of the near future technologies.

However, even with all these positive aspects cited before, reconfigurable architectures are still not largely used. The major problem precluding their usage is the necessity of special tools and compilers, modifying in somehow the source or binary code. As the old X86 ISA has been showing, keeping legacy binary code reuse and traditional programming paradigms are key factors to reduce the design cycle, allowing one to deploy the product as soon as possible on the market.

Based on all these facts, our work proposes the use of a technique called Dynamic Instruction Merging, which is a new binary translation approach implemented in hardware, used to detect and transform sequences of instructions at run time to be executed on a reconfigurable array, in a totally transparent process: there is no necessity of changing the code before its execution at all.

The employed array is coarse-grained and tightly coupled to the processor, composed of simple functional units and multiplexers. Therefore, it is not limited to the complexity of fine-grain configurations, making possible its implementation in any future technology, not just in FPGAs. Consequently, we can take all the advantages of the reconfigurable systems cited before, maintaining independence of technology and binary code reuse.

In this work we show some results concerning the potential of using such technique, demonstrating the binary translation algorithm, the structure of the reconfigurable hardware and how they interact with each other. Besides presenting the performance improvements and area overhead, we also compare our technique against a superscalar processor based on MIPS R10000.

This paper is organized as follows. Section 2 shows a review of the existing reconfigurable processors, some other approaches regarding dynamic translation of instructions and what is our contribution considering the whole context. Section 3 demonstrates the system, looking at the structure of the reconfigurable array and the algorithm itself. Section 4 presents the simulation environment and results. Finally, the last section draws conclusions and introduces future work.

Related Work

Reconfigurable Architectures

The well known ASIP circuits have specialized hardware that accelerates the execution of the applications they were designed for. A system with reconfigurable capabilities would have almost the same benefit without having to commit the hardware into silicon. A reconfigurable processor can be adapted after design, in the same way programmable processors can adapt to application changes. That is why reconfigurable systems have already shown to be very effective, implementing some parts of the software in a hardware reconfigurable logic, as shown in Figure 1. Huge software speedups [4] as well as a reduction in system energy have been achieved [5].

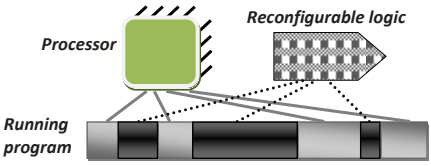

Fig. 1. An example of a reconfigurable system

Reconfigurable systems can be classified in different ways and aspects, considering coupling, granularity and instructions type [7]. A large range of systems with reconfigurable logic has already been proposed. For instance, processors like Chimaera [8], have a tightly coupled reconfigurable array in the processor core. The array is, in fact, an additional functional unit in the processor pipeline, sharing the same resources of the other units.

Reconfigurable fabric has also been applied in other levels of the architecture, imposing radical changes to the programming paradigm, involving the development of new compilers and tools. Putting this concept to the edge, an example of total dataflow architecture is the Wavescalar processor [11].

Binary Translation

The concept of binary translation (BT), illustrated in Figure 2, [12] is very ample and can be applied in various levels. BT is based on a system, which can be implemented in hardware or software, responsible for monitoring the running program. After the analysis, some transformation is done in the code, with the purpose of adapt an existing binary to be executed in a specific ISA, to provide means to enhance the performance or even both.

Fig. 2. The Binary Translation (BT) process

Existing optimizations include dynamic recompilation and caching of previous binary translation results. For instance, the Daisy architecture is based on a VLIW processor that uses binary translation at runtime to better exploit the ILP of the application [13]. One of the advantages of using this technique is that this process is transparent, since there is no need for any modifications in the binary code. Consequently, it requires no extra designer effort and causes no disruption to the standard tool flow used during the software development.

Reuse of Instructions

The idea of trace reuse is based on the principle of instruction repetition [14]. This principle relies on the idea that instructions with the same operands will be repeated a large number of times during the execution of a program. Hence, instead of executing the instruction again using an ordinary functional unit, the result of this instruction is fetched from a special memory.

Trace reuse is based on an input and an output context. For a given sequence of instructions, the context of the first instruction of this sequence is saved. The output context, in turn, is the set of results of all last instruction of this sequence. A context is composed by the program counter, registers and memory addresses. Each time that an instruction with the same input context previously found is executed again, the processor state is updated with the output context, avoiding the execution of all instructions that compose that trace. A special memory, called Reuse Trace Memory (RTM), is used for storing the values. Figure 3 summarizes this process.

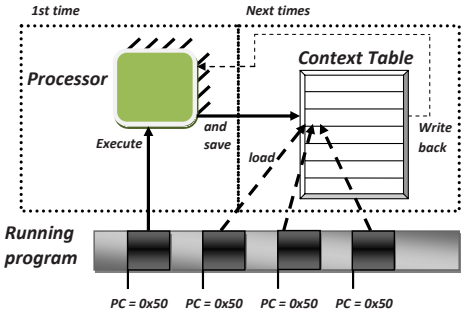

Fig. 3. The trace reuse technique

However, the context and trace sizes usually become huge, limiting the field of action of such approach, and increasing the complexity of the reuse detection algorithm. Good results are achieved just when using very optimistic assumptions, such as one cycle per trace reuse and the use of huge Reuse Trace Memories, not feasible even in future technologies because of power issues. The memory size grows

too fast mainly because identical sequences of instructions, but with different contexts (as different input operands), must occupy different slots in this special memory.

Dynamic Detection and Reconfiguration

Trying to unify some of these ideas, Stitt et al. [15] presented the first studies about the benefits and feasibility of dynamic partitioning using reconfigurable logic, producing good results for a number of popular embedded system benchmarks. The structure of this approach, called warp processing, is a SOC. It is composed by a microprocessor to execute the software, another microprocessor where the CAD algorithm runs, a dedicated memory and an FPGA. Firstly, the microprocessor executes the binary, and a profiler monitors the instructions in order to detect critical regions. After that, the CAD software decompiles it to a control data flow graph, make the synthesis and maps the circuit onto a simplified FPGA structure.

However, although the CAD system is very simplified comparing to conventional ones, it remains complex: it does decompilation, CFG analysis, place and route etc, and, according to the work, 8 MB of memory are necessary for its execution, which is still huge for nowadays on-die memories. Another issue is the use of the FPGA itself: besides area consuming, it is also power inefficient because of the excessive switches and the considerable amount of static power. As a consequence, this technique is just limited to critical parts of the software, working well just in very particular programs, such as the ones based on filters.

In [16] it is also presented a very similar reconfigurable structure used in this work: a coarse-grain array, composed by very simple functional units, tightly coupled to an ARM processor. This array is called CCA. However, in the same way of the technique above, it relies on complex graph analysis, which is performed statically with compiler help. Moreover, it does not support memory operations or shifts, and has a very small number of input and outputs allowed, limiting its field of application.

Our Approach

Our work is based on a special hardware (Dynamic Instruction Merging Machine), designed in order to detect and transform sequences of instructions to be executed on the reconfigurable hardware. This is done concurrently while the main processor fetches valid instructions. When this unit realizes that there is a certain number of instructions that are worth being executed in the array, a binary translation is applied to this sequence. This translation transforms the original sequence of instructions to a configuration of the array, which performs exactly the same function. After that, this configuration is saved in a special cache, indexed by the PC register.

The next time the saved sequence is found, the dependence analysis is no longer necessary: the processor just needs to load the configuration from the special cache and the operands from the register bank, setting the reconfigurable hardware as active functional unit. Then, the array executes the configuration with that context and writes back the results, instead of executing everything in the normal flow of the processor. Finally, the PC is updated, in order to continue the normal operation.

Depending on the size of the special cache used to keep these configurations, the increase in performance can be extended to the whole software, not being limited to loop centered applications. By transforming any sequence of opcodes into a single combinational instruction in the array one can achieve great gains, since less access to program memory and less iterations on the datapath are required.

In a certain way, the approach saves the dependence information of the sequences of instructions, avoiding performing the same job for the same sequence of instructions as superscalar processors do. It is interesting to point out that almost half of the number of pipeline stages of the Pentium IV processor is related to dependence analysis [3]; and half of the power consumed by the core of the Alplha 21264 processor is also related to extraction of dependence information among instructions [17]. Moreover, both the DIM machine as the reconfigurable array work in parallel to the processor, bringing no delay overhead or increasing the critical path of the pipeline structure.

Comparing to the techniques cited before, our approach also takes advantage of a reconfigurable system, but a coarse grain one, so it can be implemented in any technology, not just FPGAs. Together with that, we use binary translation to avoid the need for code recompilation or the utilization of extra tools, making the optimization process totally transparent to the programmer. The algorithm for the detection and transformation of binary code is very simple, in the sense that it takes advantage of the hierarchal structure of the reconfigurable array. Hence, the use of complex on-chip CAD software or graph analyzers is not necessary, which usually makes use of another processor in the system just to perform this task.

Moreover, the proposed technique relies on the same basic idea of trace reuse, where sequences of instructions are repeated. However, it presents the advantage that just one entry in the special memory is needed for the same sequence of instructions, even when they have different contexts. This takes the pressure off from the cache system, making possible its implementation with a small memory footprint, with realistic assumptions concerning execution and accesses times, even for present days technologies. Figure 4 summarizes the technique and its similarities with the previous ones.

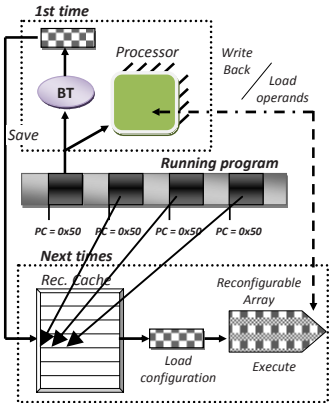

Fig. 4. The proposed approach

In the follow subsections we explain the architecture of the array, how it works together with the main processor, the detection and translation algorithm process and how the loading and execution of instructions inside the reconfigurable array are performed.

THE RECONFIGURABLE SYSTEM

Architecture of the Array

The reconfigurable unit is a dynamic coarse-grain array tightly coupled to the processor, working as another functional unit in the execution stage, using the same approach of Chimaera [8]. This way, no external accesses to the array are necessary (which in turn could increase the delay and power consumption). Furthermore, this makes the control logic simpler, diminishing the overhead required in the communication between the reconfigurable array and the rest of the system. The array is two dimensional, composed by rows and columns, where an intersection between one row and one column is represented by ordinary functional units (ALU, shifter, multiplier, etc), where each instruction is allocated. If two instructions do not have data dependence, they can be executed in parallel, in the same row.

A column is homogeneous, having always the same kind of functional unit. It is divided in groups, where each group takes a determined number of cycles to be executed, depending on the delay of each functional unit. The delay can vary depending on the technology and the way the functional unit was implemented. The detection algorithm can be adapted to different delays. For instance, according to the critical path of the processor, more sequential ALUs can be put together to be executed at the same cycle.

An overview of the general structure of the array is shown in Figure 5. Basically, there is a set of buses that receive the values from the registers. These buses will be connected to each functional unit, and a multiplexer is responsible for choosing which value will be used (Figure 5a). As can be observed, there are two multiplexers that will make the choice of which operand will be issued to the functional unit. We call them as input multiplexers. After that, there is a multiplexer for each bus line that will choose what result will continue through that line. These are the output multiplexers (Figure 5b). As some of the values of the input context or previous results generated by previous operations can be used by other functional units after it was already used, the first input of each output multiplexer is the previous result of that bus.

Note that in the simple example used in Figure 5, the first group supports up to two loads to be executed in parallel, while in the second group three simple logic/arithmetic operations are allowed. The reconfigurable array can not afford any kind of floating point operation.

Reconfiguration and Execution

As the detection for the address that will be used in the reconfiguration is done in the first stage of the pipeline, and the reconfigurable array is in the fifth stage, there are 4 cycles available between the detection and the use of the array. As one cycle is necessary to find the cache line that has the array configuration, three cycles are available for the reconfiguration, which involves the load of the values of all registers that will be used by that configuration, the load of immediate values, the configuration for the multiplexers and functional units and so on.

During the execution of the operations in the array, one issue is the load instructions. They stay in a different group in the array as shown in figure 5, and the number of columns of this group depends on the number of read ports available in the memory (which means the number of loads that can occur simultaneously). Operations that depend on the result of a load have already been allocated in the array during the detection phase, considering a cache hit as the total load delay. If a miss occurs, the whole array stops until it is resolved.

Finally, the results that need to be written back either in the memory or in the local registers are allocated in a buffer. The values will be allowed to be written back just when they are not used anymore for that configuration of the array. For instance, if there are two writes in the same register in a determined configuration, just the last one will be performed, since the first one was already consumed inside the array by other instructions.

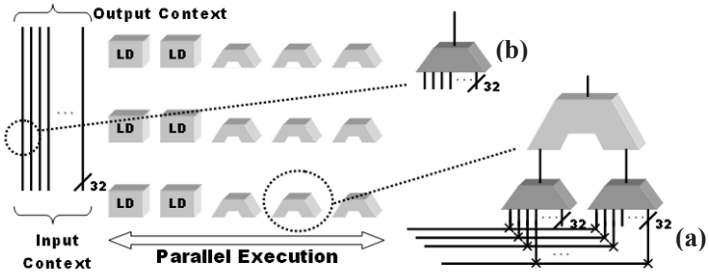

Fig. 5. The structure of the Reconfigurable Array

The Binary Translation Algorithm

Data structure

Some tables are necessary in order to perform the routing of the operands inside the reconfigurable array as well as the configuration of the functional units. Other intermediate tables are also needed, however, they are just used during the detection phase. These tables are:

Dependence table: Saves information of data dependence of each row. This table is in fact a small bitmap of 32 bits. It informs what registers in that row will be written.

Note that it is not necessary to store this information for each instruction. Summarizing the information in a bitmap for each row one can reduce the hardware necessary to check true data dependencies (RAW – read after write).

Resource Table: Stores what function each functional unit must perform.

Read Table: Informs what operand from the input context must be read. This table has two inputs, since there are two source operands for each functional unit. It is important to point out that the input context is basically an indirect table. In other words, not necessarily the first slot needs to store the value of the register R1.

Write table: This table informs what value each context slot will receive. This table is different when comparing to the read one. In the previous table the multiplexers were responsible for choosing what values from the context slots would be issued to each functional unit. This table informs what values from the whole set of the functional units that compose each row will continue in each slot of the context bus.

Context table: This table has two lines, the first one representing the input context, and will be used in the reconfiguration phase, and the second one called current table, that will be used during the detection phase. Its final state represents what values will be written when the execution of the array finishes.

How it works

To better explain the algorithm, we will start with its simplest version, considering that the array is composed just by adders. The following steps represent pipeline stages when considering the implementation in hardware.

Considering that

```
inst op_w, op_r1, op_r2
```

where `inst` is the current instruction and `op_w`, `op_r1` and `op_r2` are the target and the source operands, respectively, the follow steps are necessary.

1st) Decode the instruction, returning the target and source registers of the current instruction;

2nd) In the write table, for each row from 0 to N, verify if `op_r1` and `op_r2` exist. If any one of them or both exist in the line *S*, line *O* equals to *S + 1*. Considering a bottom-up search, the line *s* is the last one where `op_r1` or `op_r2` appears, since they may be found in more than one line. If nor `op_r1` neither `op_r2` exist in any line of this table, line *O* equals to 0.

3rd) In the resource table, search in the columns of row *O*, from left to right, if there is a resource available for use. If there exists, we call this free column as *C*, and row *R* equals to *O*. If there is no resource available in row *O*, increment the value of *O* in 1 and repeat the same operation, until finding the resource. This way, line *R* equals to *O + N*, where *N* was the number of increments necessary until finding an available resource. This resource table is also represented by a bitmap.

4th)

- Update the bitmap write table in line R with the value of `op_w`
- Update column C in row R of the resource table as busy
- Search in the current context table if there are `op_r1`, `op_r2` and `op_w`. For each one of these, if they exist, point *L1*, *L2* and *W* to `op_r1`, `op_r2` and `op_w`

respectively, and disable the correspondent write signals. If one of them does not exist in the table, the correspondent signal of write is set and the correspondent pointer is set to the next free column available.

5th)

- Depending on the step 4c, the current context table is updated.
- The initial context table is also updated, if one of the write signals concerning `op_r1` and `op_r2` are set.
- In the write table, write the value of *C* in the row R, column *W*.
- In the read table, write the values of *L1* and *L2* in line *R*, column *C* (it is important to remember that each column of this table has two slots, as explained earlier)

Summarizing the algorithm, for each incoming instruction, the first task is the verification of RAW (read after write) dependences. The source operands are compared to a bitmap of target registers of each row. If the current row and all above do not have that target register equal to any of the source operands of the current instruction, this instruction can be allocated in that row, in a column as left as possible, depending on the group, as explained before.

When this is instruction is allocated in that row, the bitmap of target registers is updated. This way, for each instruction just one bitmap per line is necessary to be analyzed. Indirectly, such technique increases the size of the window of instructions, which is one of major limiting factors of ILP, exactly due to the number of comparators that is necessary [19]. For each row there is also the information about what registers can be written back or saved to the memory. This way, it is possible to write results back that will not be used anymore in the array in parallel to the execution of other operations. Figure 6 demonstrates an example of a sequence of instructions allocated in the reconfigurable array.

The complete version of the algorithm supports functional units with different delays and functions, and the use of immediate values in the input context; handles with false data dependencies among instructions; and performs speculative execution. For the speculative execution, each operand that will be written back has a flag indicating its depth concerning speculation. When the branch is taken, it triggers the writes of these correspondent operands.

The speculative policy is one of the simplest ones, based on bimodal branch predictor. For each level of the tree of basic blocks, the counter must achieve the maximum or minimum value (indicating the way of the branch). When the counter equals to this value, the instructions corresponding to this basic block are added to that configuration of the array. The configuration is always indexed by the first PC of the whole tree. If miss speculation occurs a determined number of times, achieving the opposite value of the respective counter, that entire configuration is flushed out and another one begins, starting everything again.

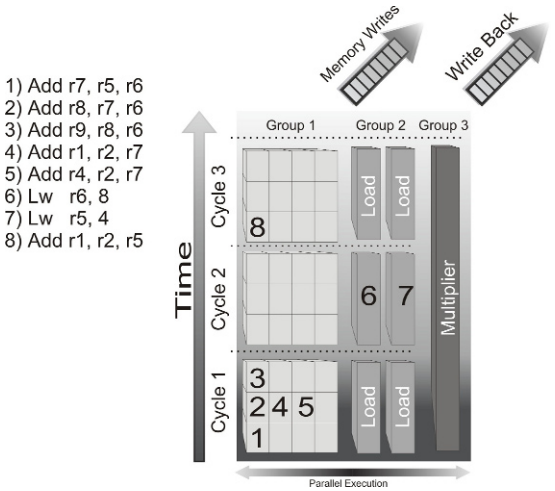

Fig. 6. An example of how a sequence of instructions is allocated inside the array

RESULTS

Performance

The Simplescalar toolset was employed for our experiments. We used the PISA instruction set, which is based on the MIPS IV ISA. Although the out-of-order simulator has some differences when comparing to the MIPS R10000 processor, we configured it to behave as close as possible to this processor. The configuration is summarized in Table 1a.

In Table 1b, we show three different configurations for the array that we used in the experiments. The last configuration was used in order to try to figure out what is the real potential of our technique. For each array configuration we also vary the size of the reconfiguration cache: 2 to 512 slots. Moreover, for each one of these configurations we evaluate the impact of doing speculation, up to three basic blocks ahead. Furthermore, we increased the cache memory in order to achieve almost no cache misses, so we can evaluate our results without the influence of it.

Table 1. Configurations

(a)

Out of Order
Fetch, decode and commit = up to 4 instructions
Register Update Unit = 16 Entries
Load/Store Queue = 16 entries
Functional Units = 2 Integer ALU, 1 multiplier, 2 memory ports
Branch Predictor = Bimodal/512 entries

(b)

	Reconfigurable Array		
	C #1	*C #2*	*C #3*
#Lines	27	54	99
#Columns	11	16	30
#ALU / line	8	8	11
#Multipliers / line	1	2	3
#Ld/st / line	2	6	8

Table 2a shows the IPC of the out-of-order processor cited before. This table can be used to compare the IPC of this processor against the IPC of the instructions that are executed inside the array, in different configurations. For each configuration, we vary the speculation: no speculation, 1 and 2 basic blocks ahead. We also change the number of slots available in the reconfigurable cache (4, 16, 64, 128 and 512). We are using a subset of the MIBENCH set [10].

Table 2. IPC in the Out-of-Order and average Basic Block size

(a)

Algorithm	IPC - Out-of-Order
Basicmath	1.43
CRC	2.13
dijkstra	1.76
Jpeg decode	1.86
patricia	1.40
qsort	1.79
sha	1.94
stringsearch	1.60
Susan Smoothing	1.64
Susan Corners	1.83
tiff2bw	1.90
tiff2rgba	1.92
tiffdither	1.56
tiffmedian	1.91

(b)

BB size
5.8751
7.9954
5.6011
6.2554
4.4255
4.6243
7.9381
4.8709
15.8098
13.4952
22.5567
13.4952
18.9188
30.686

As it is shown in Figure 7, we can achieve a higher IPC when executing instructions in the reconfigurable array in comparison to the out-of-order superscalar processor in almost all variations. However, the overall optimization when using our technique depends on how many instructions are executed in the reconfigurable logic instead of using the normal flow of the processor. Table 3 shows the overall speedup obtained when coupling the reconfigurable array to the out-of-order processor against the out-of-order without it.

The four benchmarks were chosen because they represent a very control-oriented algorithm, a dataflow one and a midterm between both, plus the CRC, which is the biggest benchmark in the set. In Table 2b the benchmarks are classified according to the average number of branches per instructions. It is important to notice that reconfigurable systems in general can just show improvements when the programs are very dataflow oriented. The proposed technique, on the other hand, can optimize control and data oriented programs, as it can be observed by the results.

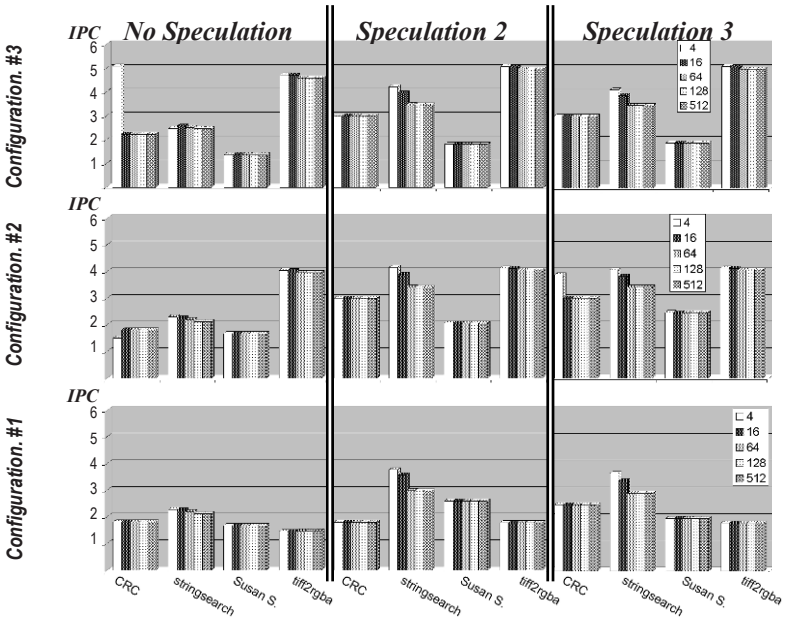

Fig. 7. IPC rate in the reconfigurable array considering different configurations and cache sizes.

Table 3. Speedups using the reconfigurable array coupled to the out-of-order processor

Algorithm	#Cycles in the Out-Of-Order	% of Speed Up - Out-of-Order coupled to array with configuration 1								
		No Speculation			Speculation 2			Speculation 3		
		4	64	256	4	64	256	4	64	256
Basicmath	111169924	5.03	13.75	17.85	3.52	14.49	21.79	3.40	15.22	23.31
CRC	399531928	-16.01	-16.03	-16.03	-5.20	-5.21	-5.21	9.03	9.03	9.03
dijkstra	31094638	-22.29	-24.31	-24.33	1.30	1.25	1.25	8.45	8.46	8.46
Jpeg decode	3942226	-9.15	-9.72	-9.77	4.63	3.24	3.29	7.11	7.45	7.61
patricia	95927575	4.41	13.30	13.72	3.99	14.42	21.52	3.26	14.22	21.96
qsort	23435690	-8.76	-11.69	-11.69	-0.58	4.18	4.18	0.37	-30.41	-30.21
sha	6800950	11.56	13.07	13.07	27.22	33.45	33.45	26.30	31.29	31.29
stringsearch	115917	16.32	20.16	21.23	28.95	35.20	35.24	28.50	35.39	35.38
S. Smoothing	15628090	-0.94	-3.22	-3.22	0.31	-0.99	-1.00	2.13	1.59	1.59
S. Corners	533870	2.16	1.79	1.79	4.40	4.29	4.28	1.13	4.29	4.28
tiff2bw	27391803	-4.24	-4.38	-4.42	0.88	0.82	0.82	-0.20	-0.20	-0.20
tiff2rgba	23796384	-10.94	-11.39	-11.40	-1.53	-1.75	-1.75	-1.19	-1.39	-1.40
tiffdither	188757828	1.48	8.88	8.92	6.65	9.34	9.41	4.47	-21.46	-23.52
tiffmedian	93254386	3.95	3.74	3.73	12.91	12.82	12.82	7.42	7.38	7.38

Algorithm	#Cycles in the Out-Of-Order	% of Speed Up - Out-of-Order coupled to array with configuration 3								
		No Speculation			Speculation 2			Speculation 3		
		4	64	256	4	64	256	4	64	256
Basicmath	111169924	5.76	19.27	26.40	4.63	19.83	30.33	4.86	20.52	32.14
CRC	399531928	3.97	3.97	3.97	8.12	8.14	8.14	20.75	20.77	20.77
dijkstra	31094638	-21.96	-20.08	-20.04	1.00	4.34	4.36	4.13	7.65	7.67
Jpeg decode	3942226	9.76	11.92	12.05	16.55	18.94	19.06	16.77	19.51	19.68
patricia	95927575	5.06	17.97	18.89	5.25	18.80	29.07	4.57	18.58	29.80
qsort	23435690	24.29	38.95	38.95	16.79	43.74	43.74	16.44	40.72	40.72
sha	6800950	22.57	25.48	25.48	39.91	48.66	48.66	41.27	50.28	50.28
stringsearch	115917	21.02	27.05	30.57	31.25	41.02	41.17	31.04	42.61	42.63
S. Smoothing	15628090	25.35	35.66	35.69	26.87	37.95	37.96	23.73	32.05	32.04
S. Corners	533870	32.69	41.44	41.44	37.53	41.44	41.45	33.89	37.13	37.12
tiff2bw	27391803	-5.65	-5.42	-5.39	19.08	19.60	19.60	24.41	25.22	25.22
tiff2rgba	23796384	57.19	57.83	57.83	58.29	59.69	59.69	47.30	48.87	48.87
tiffdither	188757828	4.33	18.15	18.30	10.73	19.33	19.57	7.95	14.31	14.60
tiffmedian	93254386	14.13	14.11	14.13	27.23	27.43	27.43	27.36	27.72	27.72

Area Evaluation

In order to give an idea of the area overhead, we implemented the hardware detection and the reconfigurable array in VHDL. The tool used was the Mentor Leonardo Spectrum [9], with the library TSMC 0.18u. As we do not have available any implementation of a superscalar processor in any Hardware Description Language, we took the data about its number of transistors from [18] and other measurements from [19]. Although this comparison will not give us exactly values, it will present realistic measurements about the implementation of our approach.

Table 4a shows how many functional units and multiplexers would be necessary to implement the configuration #1 of table 1, and what are the number of gates they take. In this same table one can also observe the number of gates taken by the Dynamic Instruction Merging hardware. In table 4b it is shown the number of bits necessary to keep one configuration in the reconfigurable cache. Note that, although 256 bits are necessary for the Write Bitmap Table, they are not counted in the final total. This table is temporary and is used just during detection. This way, there is no need to save its values in the special cache. Finally, in table 4c, the number of Bytes needed for different cache sizes is presented, depending on how much configurations they can store.

Table 4. Area evaluation

Unit	*#*	*Gates*
ALU	216	337,824
LD/ST	36	5,904
Multiplier	6	20,067
Input	510	327,420
Output	216	66,096
DIM Hardware		1,024
Total		735,223

(a)

Table	*#bits*
Write Bitmap Table	256
Resource Table	903
Reads Table	1,896
Writes Table	648
Context Start	40
Context Current	40
Immediate Table	128
Total	***3655***

(b)

#Slots	*#Bytes*
2	7,566
4	14,620
8	30,143
16	58,480
32	118,856
64	233,920
128	468,488
256	935,680

(c)

Finally, Figure 8a represents the MIPS layout with the reconfigurable array. According to [18], the total number of transistors of core in the MIPS R10000 is 2.4 million. As presented in table 4a, the array together with the hardware detection occupies 735,223 gates. We are considering that one gate (result given by the synthesis tool) is equivalent to 4 transistors, which would be the amount necessary to implement a NAND or NOR gates. This way, the reconfigurable array and DIM hardware would take 2,940,892 transistors. The area overhead is represented in Figure 6b. In this figure is also presented the area overhead concerning the reconfigurable cache, in number of different configurations supported.

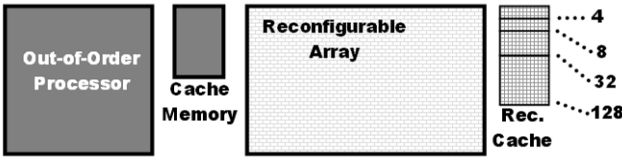

Fig. 8. Area overhead presented by the reconfigurable array and its special cache

CONCLUSIONS AND FUTURE WORK

Although there are some improvements concerning the algorithm and the structure of the reconfigurable array, this work demonstrated that it is possible to keep advantage of a reconfigurable architecture to speed up the system, in a totally transparent process and with a feasible area overhead. Using speculation in the array, we have obtained a mean speedup of up to 30% in the IPC using configuration 3, when comparing against a MIPS R10000 based superscalar processor. Now, we are working on finding the best shape for the reconfigurable array.

Another future work will be the measurement of the energy consumption of the system. Similar techniques applied to an embedded processor have already shown that such structures bring a huge energy saving [20] since, besides the fact that this technique trades sequential logic for combinational one to execute instructions, less accesses to the instruction memory are required, as well as less dependence analysis between instructions are necessary.

REFERENCES

[1] David W. Wall, “Limits of instruction-level parallelism”, In Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, p.176-188, April 08-11, 1991

[2] Sima, D., “Decisive aspects in the evolution of microprocessors”. In Proceedings of the IEEE, vol. 92, pp.1896-1926, 2004

[3] Intel Pentium 4 Homepage – http://www.intel.com/products/processor/pentium4/index.htm

[4] Venkataramani, G., Najjar, W., Kurdahi, F., Bagherzadeh, N., Bohm W., “A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. Conf. on Compiler”. In Architecture and Synthesis for Embedded Systems (CASES), 2001

[5] Stitt, G., Vahid F., “The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic”. In IEEE Design and Test of Computers, 2002

[6] Or-Bach, Z., Panel: “(when) will FPGAs kill ASICs?”, 38th Design Automation Conference, 2001.

[7] Compton, K., Hauck, S. “Reconfigurable computing: A survey of systems and software,” ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, June 2002.

[8] Hauck, S., Fry, T., Hosler, M., Kao, J.: "The Chimaera reconfigurable functional unit". In Proc. IEEE Symp. FPGAs for Custom Computing Machines, pp. 87–96, Napa Valley, CA, 1997
[9] Leonardo Spectrum, available at homepage: http://www.mentor.com
[10] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge T., Brown, R.B., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite. 4th Workshop on Workload Characterization", Austin, TX, Dec. 2001
[11] Swanson, S., Michelson, K., Schwerin, A., Oskin. M., "WaveScalar". In MICRO-36, Dec. 2003
[12] Gschwind, M., Altman, E., Sathaye, P., Ledak, Appenzeller, D.: "Dynamic and Transparent Binary Translation". In IEEE Computer, pp. 54-59, vol. 3 n. 33, 2000
[13] Ebcioglu, E. A., "DAISY: Dynamic compilation for 100% architectural compatibility". In IBM T. J. Watson Research Center - Technical Report, Yorktown Heights, NY, 1996
[14] González, A., Tubella, J., Molina, C., "Trace-Level Reuse". In Int'l Conf. on Parallel Processing, Sep. 1999
[15] Stitt, G., Lysecky, R., Vahid, F., "Dynamic Hardware/Software Partitioning: A First Approach". In Design Automation Conference, 2003
[16] N. Clark, W. Tang, and S. Mahlke, "Automatically Generating Custom Instruction Set Extensions". In Workshop on Application Specific Processors (WASP). Turkey, 2002.
[17] K.Wilcox and S.Manne, "Alpha processors: A history of power issues and a look to the future". In CoolChips Tutorial An Industrial Perspective on Low Power Processor Design in conjunction with Micro-33(1999).
[18] Yeager, K.C. "The Mips R10000 Superscalar Microprocessor,"; IEEE Micro, Apr. 1996, pp. 28-40.
[19] Burns, J.; Gaudiot, J.-L., "SMT layout overhead and scalability". In Parallel and Distributed Systems, IEEE Transactions on Parallel and Distributed Systems, pp. 142-155, Volume: 13, Issue: 2, Feb 2002
[20] Beck, A. C. S., Carro, L., "Dynamic Reconfiguration with Binary Translation: Breaking the ILP barrier with Software Compatibility", In Design Automation Conference, 2005